Contents

1 graph

1.1 Eulerian Circuit

```
/*input: line of x,y,z: vertex x and y connected by edge z
  output: edges that should be visited*/
struct graph_t {
    int nv; //nomber of vertex
    int ne; //nomber of edge
    int matrix[MAX_NV][MAX_NE];
};
graph_t G;
bool visited[MAX_NE];
//the degree of each vertex
int degree[MAX_NV];
//stack for output
stack<int> s;

void euler(int u) {
    bool flag = true;
    for (int i = 1; i <= G.nv; i++) {
        //if the degree is odd then there is no circuit
        if (degree[i] & 1) {
            flag = false;
            break;
        }
    }
    if (!flag)
        return;
    for (int e = 1; e <= G.ne; e++) {
        //if an adjacent edge is not visited
        if (!visited[e] && G.matrix[u][e]) {
            visited[e] = true;
            euler(G.matrix[u][e]);
            s.push(e);
        }
    }
}
```

1.2 Prime

```
/*input:a matrix of graph g
output:the cost of the min-covered-tree*/
void prime(){
    added[0]=true;
    int next_vertex;
    //we have n-1 vertexs to add
    int total_cost = 0;
    for (int i=1;i<n;i++){
        next_vertex=1;
        int min_cost;
        //find next vertex to be added
        for (int j=0;j<n;j++){
            if (!added[j] && distance_to_tree[j]<distance_to_tree[min]){
                next_vertex=j;
                min_cost = distance_to_tree[j];
            }
        }
        //add the new edge to the tree
        edge[i] = Edge(closed_vertex[next_vertex],next_vertex);
        total_cost += min_cost;
        distance_to_tree[next_vertex] = Max_INT;
        for (int j=0;j<n;j++){
            if (!added[j] && g[next_vertex][j]<distance_to_tree[j]){
                distance_to_tree[j]=g[next_vertex][j];
                closed_vertex[j] = next_vertex;
            }
        }

    }
}
```

1.3 Kruskal

```
struct CEdge
{
    int u;
    int v;
    int weight;

    CEdge(){}
    CEdge(int u,int v,int w):u(u),v(v),weight(w){}
};


int *root;


bool compare(CEdge a,CEdge b)
{
    return  a.weight < b.weight;//          ..
}

int Find(int x)
{
    return root[x];
}

void Union(int a,int b,int V)
{
    int root_a = Find(a);
    int root_b = Find(b);
    if(root_a != root_b)
    {
        root[b] = root_a;
        for(int i = 1 ; i <= V;i++)
            if(root[i] == root_b)
                root[i] = root_a;
    }
}

void Kruskal(int V,int E,CEdge *e)
{
    for(int i = 1 ; i <= V;i++)
        root[i] = i;

    //order by weight in edge
    sort(e,e+E,compare);

    for(int i = 0 ; i < E;i++)
        if(Find(e[i].u) != Find(e[i].v))
        {
            cout<<e[i].u<<"---"<<e[i].v<<" ";
            Union(e[i].u,e[i].v,V);
        }
    cout<<endl;
}
```

1.4 Dijkstra

```
//vs start point, prev[]:previous point, dist[] distance minimum
void dijkstra(int vs, int prev[], int dist[])
{
    int i,j,k;
    int min;
    int tmp;
    bool flag[MAX];        // flag[i]=true:already treated

    for (i = 0; i < VexNum; i++)
    {
        flag[i] = false;
        prev[i] = 0;
        dist[i] = graph[vs][i];
    }

    flag[vs] = true;
    dist[vs] = 0;

    for (i = 1; i < VexNum; i++)
    {
        //find the nearest point of the start among all points non-visited
        min = INF;
        for (j = 0; j < mVexNum; j++)
        {
            if (!flag[j] && dist[j]<min)
            {
                min = dist[j];
                k = j;
            }
        }
        flag[k] = true;

        for (j = 0; j < mVexNum; j++)
        {
            tmp = (graph[k][j]==INF ? INF : (min + graph[k][j]));
            if (!flag[j] && (tmp  < dist[j]) )
            {
                dist[j] = tmp;
                prev[j] = k;
            }
        }
    }
}
```

1.5 Bellman-For

```
/*input: matrix of the graph,origin and end
output: the shortest path from origin to end*/
bool Bellman_Ford()
{
    for(int i = 1; i <= nodenum; ++i)
        dis[i] = (i == original ? 0 : MAX);
    for(int i = 1; i <= nodenum - 1; ++i)
        for(int j = 1; j <= edgenum; ++j)
            if(dis[edge[j].v] > dis[edge[j].u] + edge[j].cost)
            {
                dis[edge[j].v] = dis[edge[j].u] + edge[j].cost;
                pre[edge[j].v] = edge[j].u;
            }
    bool flag = 1;
    //if there is a negative circuit
    for(int i = 1; i <= edgenum; ++i)
        if(dis[edge[i].v] > dis[edge[i].u] + edge[i].cost)
        {
            flag = 0;
            break;
        }
    return flag;
}
```

1.6 A*

1.7 Topological sort

```
int topologicalSort()
{
    int i,j;
    int index = 0;
    int head = 0;           //
    int rear = 0;           //
    int *queue;             //
    int *ins;               //
    char *tops;             //
    ENode *node;

    ins   = new int[VexNum];
    queue = new int[VexNum];
    tops  = new char[VexNum];
    memset(ins, 0, VexNum*sizeof(int));
    memset(queue, 0, VexNum*sizeof(int));
    memset(tops, 0, VexNum*sizeof(char));

        //calculate entering degrees
    for(i = 0; i < VexNum; i++)
    {
        node = Vexs[i].firstEdge;
        while (node != NULL)
        {
            ins[node->ivex]++;
            node = node->nextEdge;
        }
    }

    // push all points with zero entering degree
    for(i = 0; i < mVexNum; i ++)
```

```
        if(ins[i] == 0)
            queue[rear++] = i;          //

    while (head != rear)                //
    {
        j = queue[head++];              //              j
        tops[index++] = Vexs[j].data;  //                      tops       tops
        node = mVexs[j].firstEdge;      //

        //      "node"                          1
        //       1                             0
        while(node != NULL)
        {
            //          (       node ->ivex)            1
            ins[node->ivex]--;
            //                      0             "           "
            if( ins[node->ivex] == 0)
                queue[rear++] = node->ivex;  //

            node = node->nextEdge;
        }
    }

    if(index != mVexNum)
    {
        cout << "Graph␣has␣a␣cycle" << endl;
        delete queue;
        delete ins;
        delete tops;
        return 1;
    }

    //
    cout << "==␣TopSort:␣";
    for(i = 0; i < mVexNum; i ++)
        cout << tops[i] << "␣";
    cout << endl;

    delete queue;
    delete ins;
    delete tops;

    return 0;
}
```

1.8 Hungary

```
bool dfs(int u)//
{
    int v;
    for(v=0;v<vN;v++)//                       0
        {
                if(g[u][v]&&!used[v])  //                    ,        u
                {
                        used[v]=true;
                        if(link[v]==-1 || dfs(link[v])) //
                        {
                                link[v]=u;
                                return true;
                        }
                }
        }
    return false;
}
int hungary()
{
    int res=0;
    int i,u;
    memset(link,-1,sizeof(link));
    for(u=0;u<uN;u++)
    {
        memset(used,0,sizeof(used));
        if(dfs(u))
                        res++;
    }
    return res;
}
```

1.9 Strong connected component(tarjan)

```
void tarjan(int i)
{
    int j;
    DFN[i]=LOW[i]=++Dindex;
    instack[i]=true;
    Stap[++Stop]=i;
    for (edge *e=V[i];e;e=e->next)
    {
        j=e->t;
        if (!DFN[j])
        {
            tarjan(j);
            if (LOW[j]<LOW[i])
                LOW[i]=LOW[j];
        }
        else if (instack[j] && DFN[j]<LOW[i])
            LOW[i]=DFN[j];
    }
    if (DFN[i]==LOW[i])
    {
        Bcnt++;
        do
        {
            j=Stap[Stop--];
            instack[j]=false;
            Belong[j]=Bcnt;
        }
        while (j!=i);
    }
}
```

1.10 Min tree graph

2 string

2.1 KMP

2.2 Rabin Karp

2.3 Manacher's algorithm(longest palindrome)

3 flot

3.1 Max flot min cut

3.2 Min cost max flot

4 tree

4.1 Binary indexed tree

4.2 Balanced binary tree

4.3 Segment tree

5 maths

5.1 Modular Linear Equation

5.2 Chinese Remainder Theorem

5.3 Gaussian Ellimination

5.4 Bezout identity

5.5 Fast Fourier

6 geometry

6.1 Basic Library

6.2 Convex Hall